게이트레벨 연산구조를 사용한 신경망의 FPGA구현

이윤구[†] · 정 홍[††]

요 약

직접회로의 직접도 증가로 인해서 신경망을 칩으로 구현하려는 시도가 이뤄지고 있고 현재 뉴런을 모방하는 칩이 나와있는 상황이다. 하지만 이런 시도는 신경 자체를 모방하려는 것으로 아직 직접회로의 직접도를 볼 때 그런 시도가 의도하는 궁극적인 목표인 대규모의 신경망을 구현하기엔 부적합하다. 차라리 신경망의 단위를 이루는 뉴런을 구현하는 것보다는 신경망을 모방하는 시스템의 모방이 보다 적절하다. 여러 신경망이나 패턴분류 등의 신호처리에서 공통으로 필요로 하는 연산이 있다. 바로 대량의 신호를 곱하고 더하고 LUT를 읽는 연산이다. 이 연산은 신경망에서 한 층의 각 노드들로부터 그에 따른 다음 층으로의 연결들 각각을 곱해서 더해주고 시그모이드 값을 발생시키는 작업과 동일하며, 반복적이고 많은 계산량을 요구하므로 이 부분을 고속 하드웨어로 만들 경우 시스템 전체의 속도 증가를 기대할 수 있다. 그래서 여기서는 뉴런이 아닌 신경망을 구현하는데 그것은 신경망 뿐만 아닌 다른 많은 응용을 기대할 수 있는 공통적 연산부분이다. 이 연산을 앞으로 논의할 비트 분리 연산구조 방식으로 처리함으로써 실시간 병렬처리에 응용할 수 있도록 하였다.

A Neural Network's FPGA Realization using Gate Level Structure

Yunkoo Lee[†] and Hong Jeong[††]

ABSTRACT

Because of increasing number of integrated circuit, there is many tries of making chip of neural network and some chip is exit. but this is not proper because VLSI technology can't support so large hardware. So imitation of whole system of neural network is more proper. There is common procedure in signal processing as in the neural network and pattern recognition. That is multiplication of large amount of signal and reading LUT. This is identical with some operation of MLP, and need iterative and large amount of calculation, so if we make this part with hardware, overall system's velocity will be improved. So in this paper, we design neural network, not neuron which can be used to many other fields. We realize this part by following separated bits addition method, and it can be appled in the real time parallel processing.

1. 서 론

초직접 회로의 발달로 신경망을 하드웨어로 구현할 필요성이 커지고 있다. 90년대에 여러 연구소에서 부분적으로 신경망칩을 구현해보기도 하고 신경망의 일부를 하드웨어로 구현해 보기도 하였는데 근본적으로는 소프트웨어 구현이 이 분야 연구의 주류였다. 한국통신에서 개발한 아날로그 신경망 칩[1-7]이나 전자통신연구소에서 개발한 디지털 신경망칩[8], 그리고 인텔 등 많은 회사에서 개발한 신경망 칩들[9-10]은 신경자체를 모델화 하려는 시도였으며 대규모 신경망을 구성하기에는 부적합하였기 때문에 널리 보급되기에 문제가 있었다. 그리고 op-amp를 활용한 신경망 칩 등은 확장성 등에 있어서 문제가 있는 등 한계를 가지고 있다. 이런 문제들은 지금까지의 초집적 기술이 아직 신경망을 구현하기에는 적

이 연구는 문화진흥부와 BK사업단의 지원을 받음. 또한 정부 과학기술부에 의해 지원되는 뇌과학 연구 프로그램에 의해서 후원되고 있음.

† 정회원, 포항공대학교 석사과정
†† 포항공대학교 부교수

합하지 않았기 때문이라 여겨진다. 그러나 90년대 후반에 들어서 대규모의 집적회로가 점점 상용화되고 있어, 아직은 규모가 충분치 못하고 가격이 비싸 대중화되기에는 멀었지만, 연구실차원에서의 연구는 가능해지고 있다. 곧 신경망 기술에 있어서 하드웨어의 설계가 필요해지고 있는 것이다. 그런데 신경세포의 단위로 하드웨어를 구현하고 이를 엮어서 대규모의 신경망을 구현하는 이전의 방법들은 비효율적일 뿐만 아니라 집적기술이 발달한다해도 어려운 일로 사료된다. 이전 방법으로는 칩당 O(10)정도의 뉴런을 구현할 수 있었을 뿐이었다. 그렇다면 신경망 시스템을 모방하는 하드웨어적인 다른 방법을 고려하는 것이 옳을 것이다. 응용이 가능하려면 신경망 자체를 에뮬레이션 하는 방향으로 가야 하는 것이다. 곧 전기적 하드웨어의 특성이 생물의 세포에서 보여지는 특성과 구분이 된다는 점을 감안해야 한다. 그래서 이번 연구에서는 여러 신경망이나 패턴분류 등의 신호처리에서 공통으로 필요로 하는, 대량의 신호를 곱하고 더하고 변환하는 연산을 칩을 통해 구현함으로써 신경망 전체에서 계산량이 많은 부분들을 하드웨어로 수행하였다.

최근에는 FPGA의 집적도가 10만 게이트 이상이 되고 가격도 많이 내린 상태이다. 물론 칩하나의 가격이 PC 한대의 가격에 필적할 정도이지만 순수 연구차원에서 사용할 수 있게 되었다. 이 정도면 작은 규모이지만 응용이 가능한 신경망을 구현할 수 있을 것이다. 현재 신경망을 FPGA구조와 관련해 연구한 결과는 거의 없다. FPGA가 계속 발전하고 있는 최신기술이기 때문이다. FPGA의 장점은 하드웨어구조를 프로그램으로 결정할 수 있다는 것이다. 그리고 FPGA의 VHDL코드를 이용해 FPGA구조에서 ASIC구조로 하드웨어를 변환할 수 있는 CAD TOOL들이 계속해서 계발되고 있어 FPGA의 ASIC화가 용이하게 되고 있다. 그래서 이번 연구에서는 FPGA를 사용한 신경망의 에뮬레이션적 구현을 하게 되었다. 구체적으로는 먼저 신경망 패턴인식 신호처리에 공통적으로 필요한 MUL연산부를 게이트레벨로 구현하기 위한 하드웨어 구조를 연구하였으며 이 하드웨어 구조는 FPGA를 사용하여 구현하였다. 그리고 이칩을 장착한 PCI보드를 개발하고 이 시스템을 기반으로 한 응용 소프트웨어를 개발하였다. PC에서는 이를 가지고 신경망의 에뮬레이션을 실시하였는데 하드웨어의 LUT등의 특성으로 인해 MLP를 이용한 간단한 문자인식을 해보았다. 결과적으로 MLP system의 일부 연산과정들을 구현한 것이다.

곧 다음에는 이런 MLP의 전반적인 구조와 함께 내부적으로 어떠한 요소들이 칩안에서 구현되었는지에 대해 논할 것이다[11]. 그리고 이런 연산과정의 구현을 위한 3가지의 비트분리구조를 위한 하드웨어 구조를 제시하고 이중에서 구현된 아키텍쳐에 대해서 상세설명을 한 후에 실험 결과를 제시하겠다.

2. 신경망 모델

여기서의 신경망 모델은 구체적으로 MLP를 의미한다. 칩은 다른 연산과정에도 사용될 수 있으나 이번 연구에서는 앞서 설명하였듯이 하드웨어 안에서 Look up table을 지원하기 때문에 MLP를 모델로 하였다.

그림 1은 MLP-system의 전체적인 모습이다. 처음이 input layer이고 다음이 은닉층과 output layer이다.

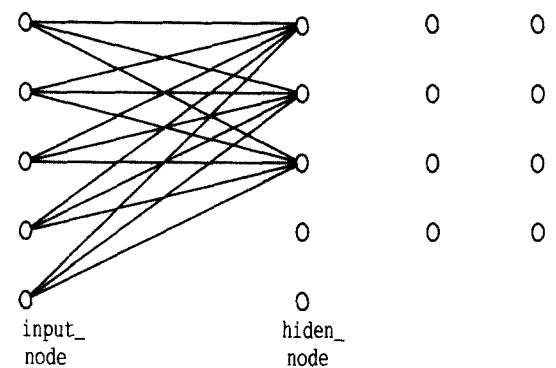



그림 1. Multi layer perceptron

각층 사이의 연결은 시냅스의 연결강도를 의미한다. 이 신경망 모델에서 좀더 세분화된 부분을 찾아볼 수 있는데 그림 2가 그것이다.

이때 forward process에서의 $Y(n)$은 다음과 같은 수식으로 표현된다.

$$Y(n) = G(\sum_{k=0}^{k-1} C(k) \cdot X(n-k)) \quad (1)$$

여기서 $G()$은 시그모이드 함수, $X(n)$은 입력노드, $C(k)$는 연결강도, $Y(n)$은 은닉노드이다. 이 부

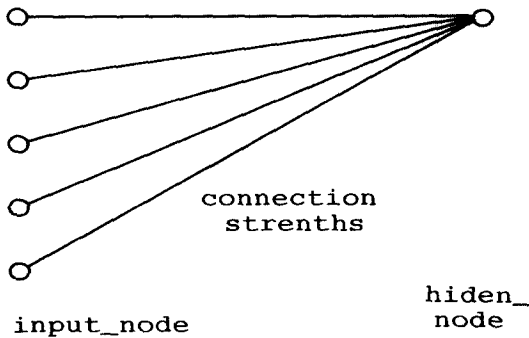



그림 2. Multi layer perceptron 일부

분을 칩에서 연산하게 된다.

전반적인 시스템은 신경망 에뮬레이터 소프트웨어(MLP), PCI 드라이버, 신경망칩으로 구성된다. 신경망칩은 신경의 기본계산을 하며 PCI드라이버는 신호를 이 칩에 공급하고 결과를 넘겨 받는다. 에뮬레이터는 신호를 신경망칩이 처리할 수 있도록 가공해 공급하고 결과를 받아 처리한다. 아울러 신경망칩을 이용해 MLP, recurrent net, Hopfield net 등 여러 종류의 신경망의 기능을 갖도록 한다. 물론 이번 연구에서는 위에서 설명한 MLP의 기능을 가지도록 하였다.

3. 하드웨어 구조

기본적으로 신경망에서 곱과 합과 LUT를 하드웨어로 구현한다. 곧 $A \cdot B(n)$(A: node의 값으로 이뤄진 행렬, $B(n)$: 연결 value로 이뤄진 행렬)을 구하고 LUT를 읽는 것이다. 이렇게 구해진 값은 따로 하위의 레지스터에 저장한 후에 읽게 된다. 곧 기본적으로 다음의 수식을 계산하는 것이다.

$$Y(n) = \sum_{k=0}^{k-1} C(k) \cdot X(n-k). \qquad (2)$$

다음 장에서는 칩에 대한 구조의 제안으로써 Sign Adder Architecture,(SAA), Adder Sign architecture(ASA), Bit Addition Architecture(BAA)의 3가지 architecture[12-14]를 제시하도록 하겠다.

3.1 The Sign Adder Architecture(*SAA*)

식 1의 수식을 전개해서 다음의 수식을 얻을 수 있다.

$$\sum_{l=0}^{L-2} 2^l \sum_{k=0}^{w-1} [\delta(x_{L-1}(k)) - \delta(x_{L-1}(k)-1)] \cdot a(k) x_l(k). \qquad (3)$$

이 수식과 연관된 그림이 그림 3에 나온다.

먼저 $a(k)$의 2의 보수를 구하고서 이를 레지스터에 저장한다. 만약 $x_{l(k)}$이 0이면 더할 필요가 없다. 하지만 1이 될 경우에는 $a(k)$이 $x(k)$의 부호에 따라서 더해지거나 빼진다, *i.e.* $x_{L-1}(k)$. 결과적으로 $x_{L-1}(k)$이 0이면 $a(k)$이 더해지고 1이면 레지스터에 저장된 $-a(k)$이 더해지게 된다.

다음부터는 이런 구조를 *Sign Adder Architecture*(SAA)라 부르겠다. 이 구조는 그림 3에 나타나 있다. 이 구조의 특징은 덧셈기에 들어가는 아규먼트(argument)가 부호비트에 의해서 선택된다는 것이다. 레지스터의 요구사항은 $(L+2M)w$ flip-flop이다.

sig ı.ıl bits에 의해서 계수를 선택하는 switching network는 $4wM$ 게이트로 이뤄져 있다.

덧셈기 트리는 Carry Propagation Array (CPA), Carry Look Ahead (CLA) 또는 Carry Save Ahead (CSA)와 같은 방식으로 구현될 수 있다. 그 중에서 CSA는 병렬성과 pipelining 특성에서 우월한 성능을 가지고 있다. CS ⌈ $\log_{1.5} frac w 2$ ⌉ A 트리로 구현될 시에 wM bit로 이뤄진 입력은 evel의 트리를 통

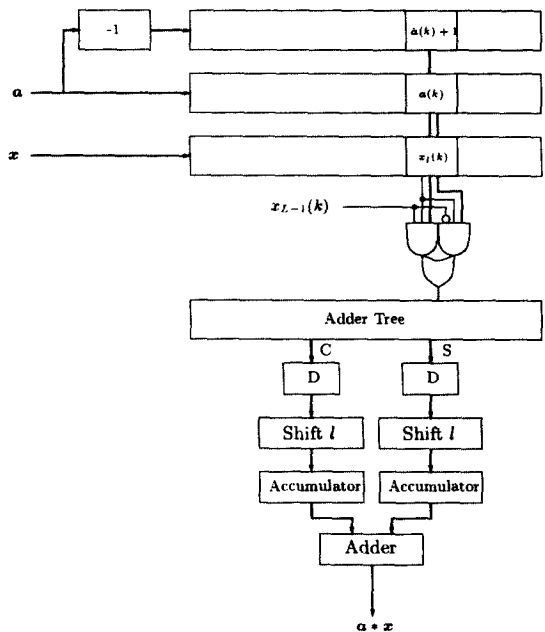



그림 3. SAA 구조

과하게 된다. 덧셈기는 $(M+L)+\lceil \log_2(p+1) \rceil$ 의 길이를 가져야 한다.

덧셈기와 누산기는 pipeline 레지스터 D에 의해서 분리되 있기 때문에 시스템의 속도는 덧셈기 트리와 누산기의 propagation delay에 의해서 결정된다. data의 분량이 크기 때문에 누산 시간은 덧셈기 트리의 propagation delay를 훨씬 초과할 것이다. 다음에 propagation delay를 줄인 구조를 설명하겠다.

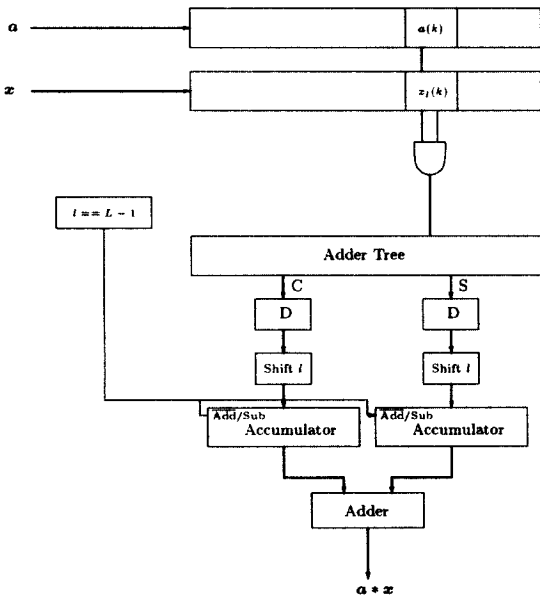
3.2 The Adder Sign Architecture(*ASA*)

SAA를 대신할 다른 대안적 구조가 있다. 수식 1은 다음과 같이 쓸 수도 있다.

$$\sum_{l=0}^{L-1} 2^{l}[1-2\delta(l-L+1)] \cdot [\sum_{k=0}^{w-1} a(k)x_l(k)]. \qquad (4)$$

이 수식과 연관된 그림이 그림 4에 나온다.

그림 4에서 특징적인 것은 부호 term이 sum of product term에 대해서 외부적이란 것이다. 1이 $L-1$에 도달할 때까지 부호는 필요하지 않다. 이런 성질은 그림 4처럼 회로를 더욱 간단하게 만든다. 이런 형식의 구조를 *Adder Sign architecture*(ASA)이라 부른다.

그림 4의 구조에서 부호는 덧셈기 트리의 끝에서 필요하다. SAA와는 달리 이 방법은 $-a(k)$를 필요로 하지 않는다. 곧 레지스터를 절약하고 동시에 2의 보수를 만들어내는 회로가 불필요해진다. 또한 덧셈기 트리에 입력을 선택하는 switching 회로가 간단해진다.

단지 $l=L-1$일 때만 덧셈기 트리의 출력이 반전되면 된다.

사실 지금의 구조는 이런 사항들을 제외하고 덧셈기 트리의 구조와 출력 덧셈기의 구조가 SAA와 동일하다. 다음에 이와는 다른 구조를 보이겠다.

SAA와 비교해서 ASA는 Mw bit를 절약하는 $(L+M)w$ bits를 가진 2개의 레지스터가 필요할 뿐이다. 또한 덧셈기 트리의 입력을 위한 switching circuit은 3wM gate가 적은 wM gate가 필요하다.

3.3 Bit Addition Architecture(*BAA*)

SAA와 ASA와 달리 정수 합의 형태를 벗어나서 bit 합의 형태를 사용하여 회로를 크게 간소화 하였다. 수식 1을 다음과 같이 전개할 수 있다.

$$\sum_{l=0}^{L-1}\sum_{m=0}^{M-1} 2^{l+m}[1-2\delta(l-L+1)][1-2\delta(m-M+1)] \cdot [\sum_{k=0}^{w-1} a_{m}(k)x_l(k)]. \qquad (5)$$

이 수식과 연관된 그림이 그림 5에 나온다.

사실 ASA의 핵심 idea가 완전한 bit level로 확장

그림 4. ASA 구조

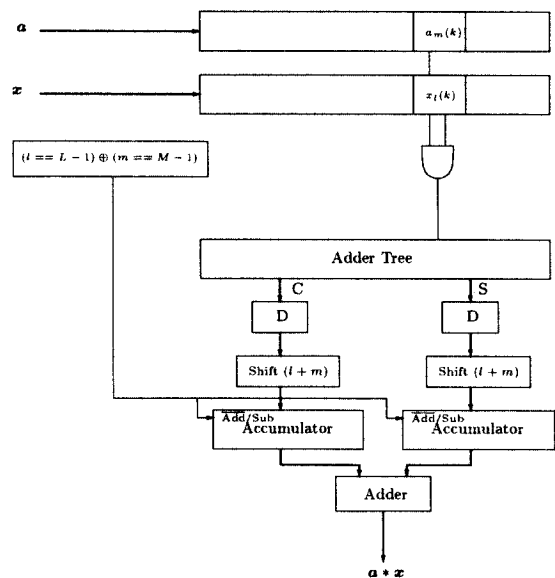



그림 5. BAA 구조

된 것이다. 부호는 $l=L-1$ or $m=M-1$(excluding the case when $l=L-1$ and $m=M-1$)에서 필요하다.

SAA와 ASA와 달리 덧셈기 트리는 정수 대신에 비트만을 받아들인다. 결과적으로 덧셈기의 수가 줄어들게 된다. 덧셈기의 깊이는 아직 $\log_{1.5}\frac{w}{2}$이다.

3.4 면적-시간 그리고 계산적 복잡도

이제 앞에서 설명한 3가지 구조에서 계산기 트리와 누산기의 계산상의 복잡도에 대해 설명하겠다. 누산기가 덧셈기 트리에 연결된다면 system clock은 단지 adder 트리에 의해서 결정된다. 또한 모든 구조는 동일한 누산기와 output 덧셈기를 지니게 된다. 3가지 구조를 비교한기 위해서 단지 계수 레지스터들, adder trees, decoding circuitry에 대해서 고려하겠다.

N_r은 레지스터에서 비트당 gate의 수를 의미한다. 일반적으로 $N_r=6\sim7$ gate가 된다.

N_a는 full adder를 위한 gate의 수를 지칭한다. 이 값은 일반적으로 $N_a=7\sim8$이 된다.

많은 공간을 차지하는 구조 중에서 주요한 요소중의 하나는 덧셈기 트리이다. CSA에서 구현 시에 각각의 덧셈기는 3개의 data를 받아들이고 2개의 output을 생성한다. 이는 계산을 복잡하게 만드는데 예를 들면 SAA, ASA, BAA를 위한 깊이 D는 다음과 같다.

$$D=\lceil \log_{1.5}\frac{w}{2}\rceil. \tag{6}$$

여기서 $w\geq2$. 또한 full adder의 수는 다음과 같다.

$$\sum_{n=1}^{D}(M+(n-1))(\frac{2}{3})^{n}w = w(2(M-1)(1 -(\frac{2}{3})^{D})+6(1-(\frac{2}{3})^{D})-3D(\frac{2}{3})^{D+1}) \quad (7)(8)$$

$w\gg2$에 대해서 quantity는 약 $2wM$이 된다. 여기서 BAA에 대해서 $M=1$. 만약 $D=\log_{1.5}\frac{w}{2}$를 가정한다면 이 양은 대략적으로 $w\gg2$에 대해서 $2wM$이 된다.

만약 모든 full adder가 pipelined 된다면 pipeline 레지스터의 수는 다음과 같다.

$$\sum_{n=1}^{D}2(M+n)(\frac{2}{3})^{n}w = 2w(2M(1-(\frac{2}{3})^{D}) +6(1-(\frac{2}{3})^{D})-3D(\frac{2}{3})^{D+1}). \quad (9)\ (10)$$

여기서 BAA에 대해 $M=1$. $D=\log_{1.5}\frac{w}{2}$에 대한 대략적인 값은 $4wM$. 식 8과 식 10은 간략한 식으로 변형될 수 있는데 곧 $D\approx\log_{1.5}\frac{w}{2}$을 설정할 수 있다. 이 경우, 계수 레지스터, switching circuit, 덧셈기 트리, pipeline 레지스터에 대한 gate 수는 표 1에 나타나 있다.

표에서 전체 면적은 4개의 주요한 요소의 합이다. 이 결과는 표 2에 나타났다.

여기서 BAA는 다른 것들보다 M배 적은 gate를 취하고 있다.

표 1. Area of major elements

Quantity	SAA	ASA	BAA
Coefficient Registers	$(L+2M)wN_r$	$(L+M)wN_r$	$(L+M)wN_r$
Switching circuit	$4wM$	wM	w
Adder Tree	$[2w(M+2)$ $-4\log_{1.5}\frac{w}{2}]N_a$	$[2w(M+2)$ $-4\log_{1.5}\frac{w}{2}]N_a$	$6w$ $-4\log_{1.5}\frac{w}{2}N_a$
Pipeline registers	$[4(w-2)(M+3)$ $-8\log_{1.5}\frac{w}{2}]N_r$	$[4(w-2)(M+3)$ $-8\log_{1.5}\frac{w}{2}]N_r$	$[16(w-2)$ $-8\log_{1.5}\frac{w}{2}]N_r$

표 2. Total area in gates

quantity	SAA	ASA	BAA
Total area without pipelining	$(L+2M)wN_r+4wM$ $+[2w(M+2)$ $-4\log_{1.5}\frac{w}{2}]N_a$	$(L+M)wN_r+wM$ $+[2w(M+2)$ $-4\log_{1.5}\frac{w}{2}]N_a$	$(L+M)wN_r+w$ $+6w$ $-4\log_{1.5}\frac{w}{2}N_a$
Total area with pipelining	$(L+2M)wN_r+4wM$ $+[2w(M+2)$ $-4\log_{1.5}\frac{w}{2}]N_a$ $+[4(w-2)(M+3)$ $-8\log_{1.5}\frac{w}{2}]N_r$	$(L+M)wN_r+wM$ $+[2w(M+2)$ $-4\log_{1.5}\frac{w}{2}]N_a$ $+[4(w-2)(M+3)$ $-8\log_{1.5}\frac{w}{2}]N_r$	$(L+M)wN_r+w$ $+6w$ $-4\log_{1.5}\frac{w}{2}N_a$ $+[16(w-2)$ $-8\log_{1.5}\frac{w}{2}]N_r$

pipeline된 adder에 대해서, 그림 6에 필터 탭 수 w와 면적 사이에 보여지는 상관관계를 그렸다.

x축은 탭 수를 나타낸다. y에 나타나는 면적은 gate 수 또는 bit수를 나타낸다. 계산상의 편의를 위해서 $L=M=8$, $N_r=N_a=7$을 가정하였다.

이 그림은 면적의 관점에서는 bit 덧셈 방식이 가장 좋은 것임을 보여주고 있다. 그러나 processing time에서는 M배의 시간이 더 걸린다. 그림에서 곡선은 하위 bound임에 유의해야 한다. 실제 구현에서는 output adder, control logic등의 추가적인 회로가 필요하기 때문이다.

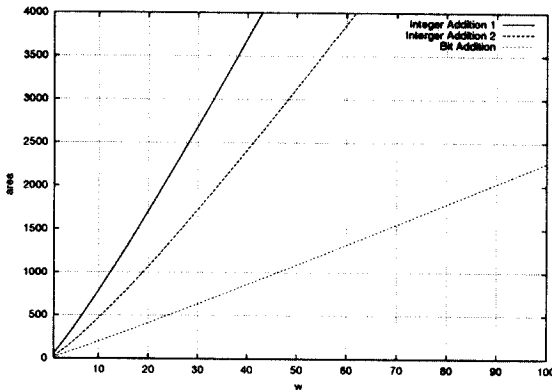



그림 6. Area vs. the number of filter taps w

t_f는 full adder의 delay를 나타낸다고 하자. 우리는 pipeline의 유무 각각에 대해서 클럭 주기를 알아야 한다. 또한 하나의 sample output이 가능한 시간을 알아야 한다. 모든 이러한 양들은 표 3에 나타나 있다.

BAA는 다른 것들보다 M배 느리다. 그러나 면적과 시간의 곱은 다른 것들보다 눈에 띄게 작다. 특히 BAA는 작은 양의 레지스터와 switching 회로, 덧셈기 트리가 들어간다.

주어진 공간에 대해서 필터 탭의 수 w는 적어도 M배 이상 길 수 있다. 이 경우, 클럭 주 $\log_{1.5}\frac{w}{2}$ 에 비례해서 증가한다. 그러나 덧셈기 트리를 pipelining 함으로써 줄여질 수 있다. 늘어나는 탭수는 많은 경우 data 전송의 bottle neck인 bus접속 수를 감소시킨다.

이 multi-chip 시스템들에서 면적과 throughput은 거의 선형적으로 증가한다.

이때 단지 pipelining buffer가 추가적으로 필요할 뿐이다.

표 3. Time comparison of the three architectures.

Quantity	SAA	ASA	BAA
Clock period without pipelining	$t_f \log_{1.5}\frac{w}{2}$	$t_f \log_{1.5}\frac{w}{2}$	$t_f \log_{1.5}\frac{w}{2}$
Output period without pipelining	$t_f Lp/w \log_{1.5}\frac{w}{2}$	$t_f Lp/w \log_{1.5}\frac{w}{2}$	$t_f LMp/w \log_{1.5} w$
Clock period with pipelining	t_f	t_f	t_f
Output period with pipelining	$t_f Lp/w$	$t_f Lp/w$	$t_f LMp/w$

3.5 병렬적 덧셈기 트리와 누산기

덧셈기 트리의 경로가 길기 때문에 시스템 클럭의 속도는 사실상 덧셈기 트리에서 좌우한다. 그러므로 sum과 carry를 트리에서 분리하고 이를 말단에서 더해주는 작업을 필요로 한다. 이런 CSA(Carry Save Adder)방식은 완전한 병렬적 디자인이다. w 비트의 합에서 트리는 $\log_{1.5}\frac{w}{2}$ 배에 비례하는 시간을 소요하게 된다. 때문에 한 클럭의 주기가 이에 따라서 늦어지게 된다. 만약 트리 안에 레지스터를 삽입하면 좀더 빠른 클럭 주기를 구현할 수 있는데 이에 따른 클럭의 손해는 감수해야 한다.

일상적으로 많이 쓰는 carry-look-ahead 덧셈기의 경우엔 누산기가 word 길이에 비례하는 시간을 쓰게 되어있다. over flow를 방지하기 위해서 누산기는 $M+L+w-1$의 길이를 가져야 하며 그 만큼의 시간이 필요하게 된다. 게다가 이 누산이 loop에 있으므로 덧셈기 트리의 매 작동 후에 작동되어야 한다. 여기서 사용한 방법은 누산기를 모든루프 밖으로 옮기는 것이다. 그래서 누산은 말단에서 한번만 실행되게 된다.

다음의 덧셈을 보자.

$$s = \sum_{n=1}^{N} a(n). \tag{11}$$

$a(n)$은 N_a 비트를 가진 입력이다. 우리가 부분합을 다음처럼 정의한다면

$$s(n) \equiv \sum_{k=1}^{n} a(k). \tag{12}$$

식 11은 다음의 합산과 같이 표현된다.

$$s(n) = s(n-1) + a(n), \quad n = 1, 2, \ldots, N. \tag{13}$$

여기서 $s(0)=0$. iteration의 끝에서 우리는 마지막 합($s=s(N)$) $s(N)$을 얻는다.

이 수식은 binary representation으로 다시 씌어질 수 있다.

$$\sum_k 2^k s_{k(n)} = \sum_k 2^k s_{k(n-1)} + \sum_k 2^k a_{k(n)}. \tag{14}$$

여기서 $s_k(n)$과 $a_k(n)$은 비트임.

이제 우리는 합이 비트체계에 의해서 표현되었음을 가정할 수 있다. 곧 $s_k(n-1)$이 $a_k(n)$와 함께 더해져서 carry 비트인 $c_k(n)$과 sum 비트인 $p_k(n)$을 생성한다. 그러므로

$$s_k(n) = p_k(n) + 2c_k(n). \tag{15}$$

방정식 14에 의해서

$$p_k(n)+2c_k(n)=p_k(n-1)+2c_k(n-1)+a_k(n). \quad (16)$$

$2c_k(n)$은 $c_k(n)$을 다음의 높은 비트 덧셈기 $k+1$에 연결함으로써 구현된다. 그러므로

$$p_k(n)+2c_k(n)=p_k(n-1)+c_{k-1}(n-1)+a_k(n). \quad (17)$$

n에 대해서 $c_0(n)=0$임. 이 수식은 $p_{k(n-1)}$, $c_{k-1}(n-1)$, $a_k(n)$을 더해서 sum 비트인 $p_k(n)$과 carry 비트인 $c_k(n)$을 내보냄을 의미하고 있다. 이 연산은 full adder에 의해서 구현된다:

$$p_k(n)=a_k(n)\oplus p_k(n-1)\oplus c_{k-1}(n-1),$$
$$c_k(n)=a_k(n)\cdot[p_k(n-1)\oplus c_{k-1}(n-1)]+p_k(n-1)\cdot c_{k-1}(n-1). \quad (18)$$

연산의 말단에서 sum과 carry를 합하는 연산이 필요하다:

$$s=p(N)+2c(N). \quad (19)$$

이 과정은 loop가 없이 단지 한번의 연산만 필요하다.

식 18과 식 19를 가지고 그림 7과 같은 회로가 구성된다. 특징적인 것은 뺄셈이다. 비트 합산의 방식과 같은 특수한 경우에는 loop의 마지막에 뺄셈이 필요하다. 이 연산은 그림에서 보여지는 것과 같이 손쉽게 하드웨어로 구현된다.

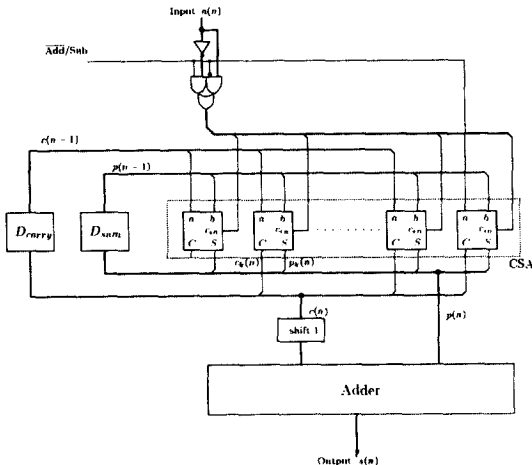



그림 7. CSA 누산기

4. 연구에서 구현된 ASA의 상세 설명

4.1 전반적인 구성

위에서 소개한 3가지 구조 중에서 ASA방식을 칩으로 구현하였다. 이는 현재 PCI에서 속도 제한을 감안할 경우 propagation delay를 이 구조가 충분히 만족하고(burst로 쓰고 읽는 것을 반복하는 것을 wait state없이 지속할 수 있음) 누산기의 구현이 SAA보다 간편하기 때문이다.

이 구조와 부합하는 식을 전개해보면 다음과 같다.

$$Y(n)=\sum_{k=0}^{p-1}C(k)\cdot X(n-k). \quad (20)$$

여기에서 $X(n)$을 다음과 같이 비트레벨로 표현할 수 있다.

$$X(n)=[\sum_{l=0}^{L-2}2^l\cdot X_l n]-2^L-1\cdot X_{L-1}(n). \quad (21)$$

이식에서 $X_l(n)$은 n번째 실수 입력데이터의 1번째 비트를 가리킨다. 곧 다음의 수식이 나오게 된다.

$$C(k)\cdot X(n-k)=[\sum_{l=0}^{L-2}2^l\cdot C(k)\cdot X_l(n-k)]-2^{L-1}\cdot X_{L-1}(n-k). \quad (22)$$

이를 대입해서 정리하면 다음과 같이 나오게 된다.

$$\begin{aligned}Y(n)&=\sum_{k=0}^{p-1}C(k)\cdot X(n-k)\\&=\sum_{k=0}^{p-1}[\sum_{l=0}^{L-2}2^l\cdot C(k)\cdot X_l(n-k)-2^{L-1}\cdot C(k)\cdot X_{L-1}(n-k)]\\&=\sum_{l=0}^{L-2}2^l\cdot\sum_{k=0}^{p-1}C(k)\cdot X_{l(n-k)}-2^{L-1}\cdot\sum_{k=0}^{p-1}C(k)\cdot X_{L-1}(n-k).\end{aligned}$$

위에서 X는 다음 그림에서 상위 레지스터에 들어가게 된다. 계수인 C는 다음 그림에서 볼 수 있듯이 계수 레지스터에 들어 간다.

시스템의 전반적인 작동은 이러하다. 누산기에 먼저 계수로서 노드의 값들을 집어 넣는다. 그리고

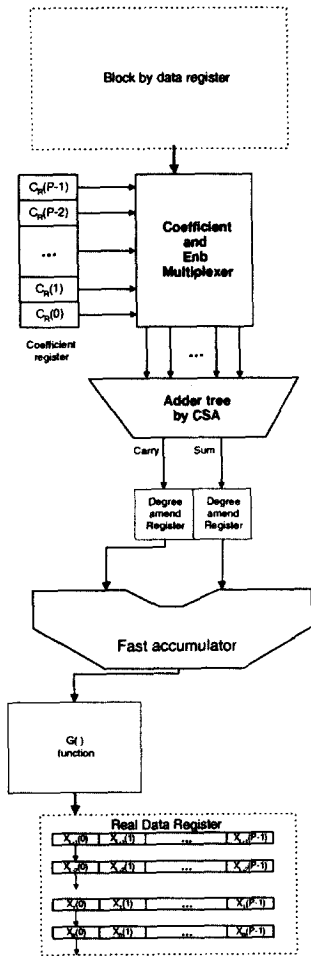



그림 8. 전체 연산 구조.

shift 레지스터에 이 노드들에 곱하게 될 값들을 로드한다. 로드가 끝나게 되면 자리수의 개수만큼의 클럭동안 이 shift 레지스터를 움직이면서 누산을 시작해서 다음 노드의 처음 값을 얻게 된다. 이를 G function을 거치고 나서 하위의 shift 레지스터에 저장한다. 그리고 나서 다시 처음에 말한 shift 레지스터에 다음 connection을 저장하고 계산해서 마찬가지로 G function을 거친 후에 하위의 레지스터에 저장한다. 이런 방식으로 계속 진행해서 하위의 레지스터를 다음 노드의 값들로 모두 채운 후에 그 값을 컴퓨터로 로드하게 된다.

4.2 레지스터들

그림 9는 계수 레지스터로써 정수 계수들을 로드

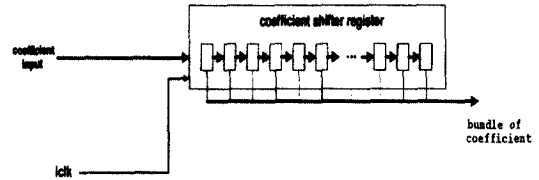



그림 9. 계수 shift 레지스터.

한다. PCI에서 오는 한클럭에 1개씩 로드하게 되어 있다. PCI로부터 오는 data가 32비트이지만 이부분을 단순화 시키기 위해서 한번에 하나의 data를 로드하게 하였다. 전체적인 cycle에서 이 부분이 차지하는 시간이 작기 때문에 속도에 있어 영향을 크게 미치지 않는다.

그림 10은 상위 레지스터의 전체적인 모습을 나타낸 것이다.

이것은 10개의 블럭을 8개 이어 붙인 것으로 내부의 상세한 모습은 그림 11에 나타나 있다.

그림 11은 계수 부분과는 달리 두개의 레지스터를 사용하였다. 먼저 상단의 레지스터는 PCI에서 오는 값을 한 PCI클럭에 4개씩 채운다. 이렇게 해서 전체를 채우면 하단의 레지스터에 이 값들을 그대로 로드한다. 이렇게 로드된 data는 계산부분에 적용되는 클럭에 동기되어 한 클럭에 하나씩 회전하게 된다. 이런 방식을 사용한 것은 데이터를 로드하는 동안에 모든 연산을 처리하기 위함이다. 곧 연산시간이 데이터의 로딩 시간보다 작으므로 데이터를 로딩하는 동안 모든 연산이 이뤄지게 된다.

4.3 계산 부분

여기서는 계산 파트를 상단에서 하단의 순서로 차례차례 설명하겠다.

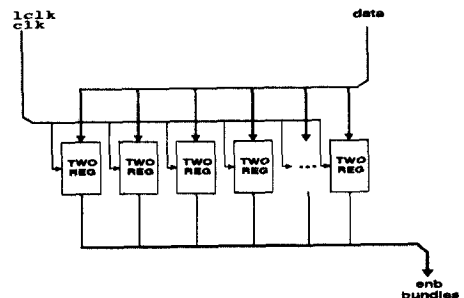



그림 10. 레지스터 블럭.

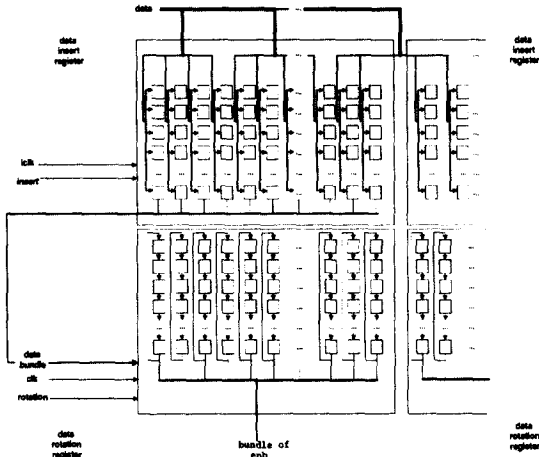



그림 11. 상단의 데이터 shift 레지스터.

그림 12는 상단의 레지스터 블럭에서 오는 enb 시그널과 계수를 곱하는 부분이다. 이 부분에서 그림 11에서 내려오는 동일한 자리수의 비트들을 계수들과 곱해서 아래 단의 CSA 트리로 들어가게 된다.

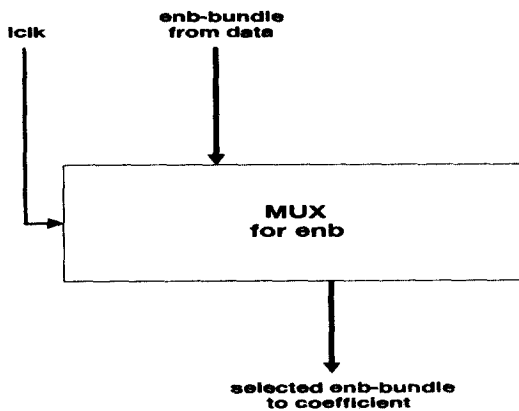



그림 12. MUX 블록.

그림 13은 CSA 트리로써 이를 사용해서 mux블럭의 비트들이 합해져 sum과 carry로 나오게 된다. 그런데 비트별로 값을 합산하기 때문에 누산기에서 누산하기 전에 자리수를 보정해 주어야 한다. 곧 이 sum과 carry는 그림 8의 전체 그림에서 볼 수 있듯이 자리수 보정 래지스터를 거치게 된다. 자리수 보정 래지스터에서는 자리수 별로(counter를 이용) 값을 보정한다. 이렇게 해서 보정된 값은 그림 14의 누산기에 들어가게 된다.

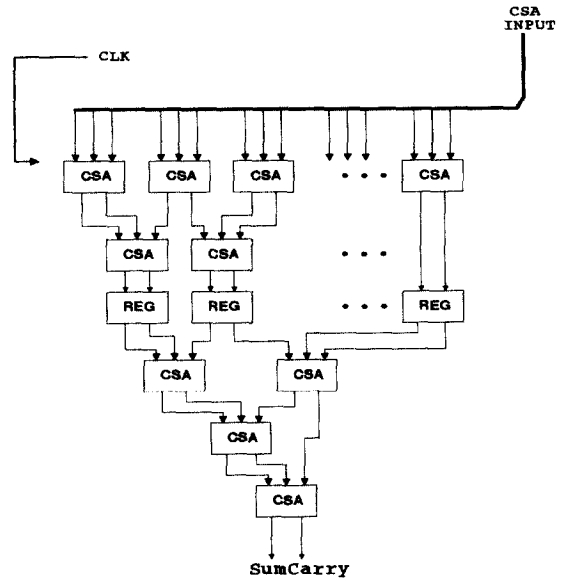



그림 13. CSA TREE.

사실 계산 부분의 클럭 속도 증가를 위해서 위의 CSA 내부에 파이프를 두단 집어 넣었다. 신호가 내려오면서 약해지기 때문에 생기는 delay path가 때문이다.

중간에 두 개의 파이프를 박으면 2개의 클럭을 손해보지만 클럭 속도가 증가해 전체적으로 속도의 증가를 가져오게 된다. 그러나 기본적인 설계는 그림 13과 같다. 그림 14는 누적 연산기와 CPA이다.

먼저 덧셈기 트리에서 오는 값들을 그림 14의 누산기에서 누산시킨다. 처음 값이 들어오면 0과 함께 들어오는 값을 합산해서 내보내는데 다음부터는 이렇게 더해진 값들과 새로 오는 값들이 합산되게 된다. 그러다가 마지막에 최상위 비트들의 합이 들어오면 그림 7에 나온 구조에 의해 뺄셈을 하게 된다. 그림 15는 누적 연산기를 좀더 상세히 보여준다. 이 부분에서 $c(n)$은 자리수 보정후 아래 부분의 덧셈기를 통해 $s(n)$과 함께 더해지게 된다. 일단 이렇게 나온 값은 PCI에서 읽기 위해서 그림 16의 shift 레지스터에 저장된다.

그림 16은 하위단의 shift 레지스터이다. 누적 연산기에서 합산된 결과가 이쪽으로 하나씩 들어오게 된다.

이런 식으로 값들이 하나씩 저장되다가 필요한 만큼 연산이 끝나면 컴퓨터에서 PCI를 통해 이쪽의 결

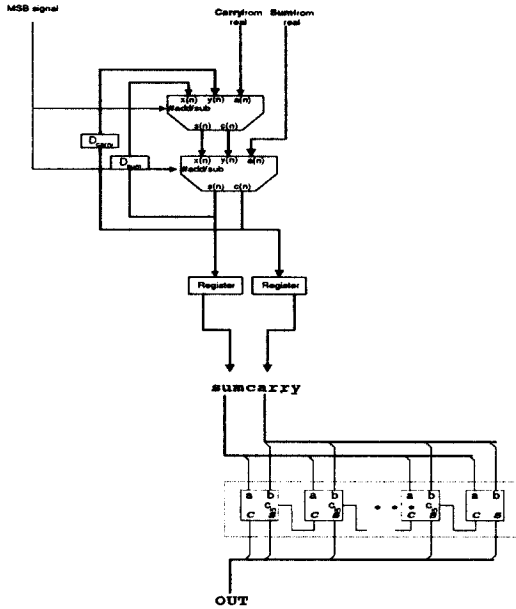



그림 14. 누적 연산기와 CPA.

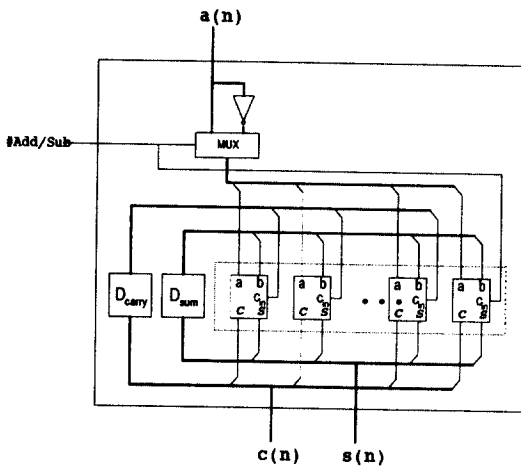



그림 15. 누적 연산기.

과를 읽게 된다. 그 후에 이부분은 바로 초기화 된다.

5. 전체 시스템 및 실험

5.1 전체적인 시스템의 모습

전반적인 시스템이 그림 17에 나타나있다.

이 시스템은 신경망 에뮬레이터 소프트웨어, PCI

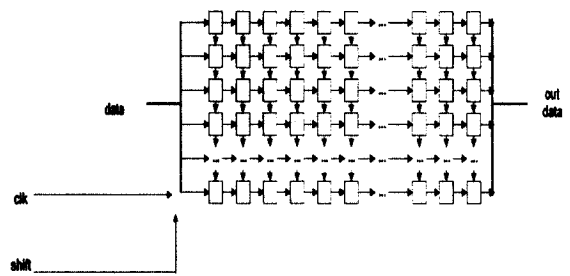



그림 16. 하단의 데이터 shift 레지스터.

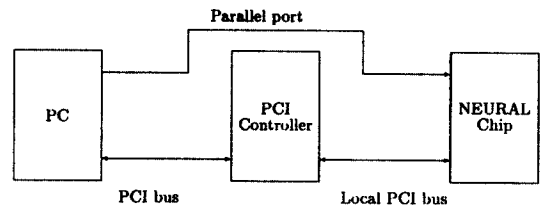



그림 17. 전체 시스템.

드라이버, 신경망칩으로 구성된다.신경망칩은 신경의 기본계산을 하며 PCI드라이버는 신호를 이 칩에 공급하고 결과를 넘겨 받는다. 에뮬레이터는 신호를 신경망칩이 처리할 수 있도록 가공해 공급하고 결과를 받아 처리한다. 아울러 신경망칩을 이용해 MLP, recurrent net, Hopfield net 등 여러 종류의 신경망의 기능을 갖도록 한다.

여기서 신경망 칩은 시프트레지스터, 연산부, 누산기, LUT, 제어부, PCI 통신부로 구성되어 있다. 시프트레지스터는 입력신호와 신경망 연결강도를 저장하고 있고 연산부는 CSA를 이용한 고속 덧셈기이며 누산기는 이 결과를 바복적으로 더하거 LUT 에서는 시그모이드값을 읽어낸다. PCI부는 이 칩을 PCI Bus와 연결하며제어부는 이 모든 시스템을 통솔하고 제어한다.

5.2 실험

간단한 문자인식을 칩을 사용해서 구현해 보았다. 10개의 알파벳을 인식하는 다층 펄셉트론에서 입력에 대해서 각 뉴론의 값들을 구하는 계산과정을 칩을 사용해서 구했는데 계수로는 입력값을 넣고 그림 11에 보이는 레지스터에는 연결 강도를 넣었다. 한번 계수레지스터를 채운 다음에 그림 11의 레지스터를 다음 층으로 넘어 가기 전까지 계속해서 채움으

로써 출력 값들을 구해 내었다. 이렇게 구해진 출력들은 칩 내부의 shift 레지스터에 저장되었다가 PCI를 통해서 읽어 들인다.

10개의 알파벳 각각은 8*8로 이뤄진 −1과 1의 조합으로 표현되었으며 곧 입력은 64개의 노드로 구성된다.

은닉 층은 하나고 44개의 펄셉트론으로 구성된다. 출력은 8개의 펄셉트론을 가지고 있다. 시그모이드 함수는 하드웨어 내부에 있는 LUT로써 대체되었는데 −1.27과 1.27 사이를 26개의 구간으로 나눈 값들을 사용하였다.

결과적으로 그림 18과 같은 학습 그래프를 얻었다.

학습 graph에서 가로축은 iteration을 나타낸다. 이에 대해서 세로축은 mean square error를 나타내고 있다. 이렇게 나온 20개의 학습 그래프들을 평균하면 그림 19와 같은 학습 그래프가 나오게 된다.

그림 19에서 interation이 반복될 수록 점점 오차가 감소하고 있다. 오차가 수렴하면서 도중에 변동

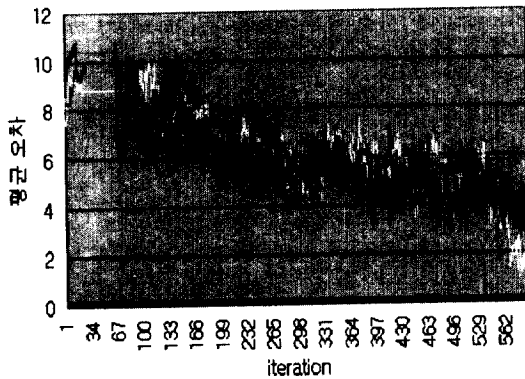



그림 18. 학습 그래프들.

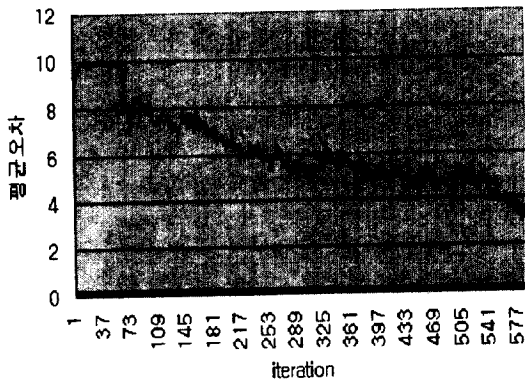



그림 19. 평균한 학습 그래프.

이 심하게 나타나고 있는데 이것은 LUT에서 나오는 값이 실제 tanh함수의 시그모이드 값이 아닌 성긴 LUT 값이기 때문으로 보인다.실제로 겉으로 보게 되는 칩의 속도는 데이터의 로딩 시간에 의해서 결정된다. 그래서 표 4에 이 시간들을 몇몇 구체적 실행 시간에 대해서 적어 보았다.

표 4에서 볼 수 있듯이 로딩하는 시간이 일정하지 않고 계속해서 변화의 양상을 보이는데 이것은 BX 칩셋에서 각 버스를 통제하는 메카니즘에 의한 것으로 보인다. 이런 식으로 500개의 데이터를 모아서 이것을 평균해 보면 표 5에서 볼 수 있는 평균 로딩 시간을 알 수 있다.

표 4. 데이터의 처리에 따른 시간.

시간(초)	CoeffTime	RegTime	ReadTime	TotalTime
	1	5	0	5
	0	5	0	6
	1	5	0	5
	0	5	0	6
	1	4	1	5
	0	5	0	5

표 5에서 볼 수 있듯이 대부분의 시간은 레지스터를 로딩하는데 걸린다. 사실 20메가의 속도로 처리하기 때문에 8클럭의 계산시간은 $8/20M = 0.4*10^{-6}$ 밖에 걸리지 않는다. 곧 여기서의 시간은 순전히 loading time인 것이다.

표 5. 평균적인 로딩 시간.

시간(초)	CoeffTime	RegTime	ReadTime	TotalTime
평균로딩시간	0.30	4.19	0.13	4.61

6. 결 과

고속이면서 대용량의 뉴럴 네트웍을 구현하기 위해서 웨이퍼 스케일 인테그레이션(WSI)나 VLSI, 디지털 회로를 사용하는 등의 방법이 연구되고 있다. WSI방식은 일반적인 VLSI방법보다 고속이면서 대용량의 회로를 구현할 수 있다는 장점이 있다.[15-16] 하지만 웨이퍼에 결점이 있는 경우에 정상적인 연산장치를 구성하는 것이 어렵게 된다. 뉴럴 네트웍이 어느 정도의 에러에 대해서 견딜 수 있다는 점은 이런 결점을 극복할 수 있게 만든다.

디지털회로는 대용량의 뉴럴 네트웍을 구현하기

위해 점점 적절한 방법이 되어가고 있다.[17] 현재 계속해서 발전하고 있는 직접기술과 통신용 FPGA의 수요가 증가하는 추세 등으로 디지털 회로 기술은 점점 발전하고 있다. 또한 디지털 회로로 구현된 시스템은 호스트 컴퓨터와 쉽게 연결될 수 있다는 장점도 가지고 있다. 설계상의 용이함도 장점이다. 하지만 아날로그보다는 트랜지스터의 효율에서 떨어진다.

그래서 이번 연구에서는 이러한 특징을 염두에 두고 아날로그와는 다른 접근을 시도해 보았다. 이전의 아날로그 회로로는 한칩에 16개 정도의 뉴론이 들어가는 수준이었다.[18] 디지털 회로로(FPGA) 구현할 경우에는 이보다 더 큰 수의 뉴런을 넣을 수 있지만 트랜지스터의 효율면에서 보면 이는 바람직하지 않은 것이다. 그래서 뉴럴네트웍의 뉴런을 구현하지 않고 뉴런들이 연결된 전체적 시스템을 부분적으로 구현하였다. 이를 통해서 여러 개의(80개까지) 뉴런을 흉내낼 수 있었다.

결과적으로 80tab의 8비트의 data를 20메가의 속도로 현재의 FPGA 장치인 flex10ka-agc599-3에서 처리하였다. 하지만 컴퓨터의 bus를 burst로 사용한다고 하여도 이 속도를 따라가지 못하였다. 이는 현재 실험을 실시한 컴퓨터의 마더보드가 인텔사의 BX 칩셋을 사용하는 관계로 burst의 효율이 떨어지고 많은 data를 칩으로 다운로드내지는 업로드해야 되기 때문으로 보인다. 사실 이 구조는 칩을 위한 design이기 때문에 이런 사항들은 크게 문제가 되지 않는다고 판단된다.

만약 데이터를 다운로드하는 속도가 빨라진다면 현재보다 빠른 속도로 데이터를 처리하게 된다. 그리고 더많은 노드가 필요하다면 같은 연산장치를 하나 더 추가하고 추가된 장치의 상위 레지스터와 계수 레지스터에 추가적인 노드 값들과 계수값들을 로드해서 계산한 후에 결과 값들을 단순히 더해 주기만 하면 된다. 보다 빠른 연산을 위해서도 장치를 더 추가하고 필요한 약간의 부가 작업을 더해 주면 된다.

곧 이번 연구에서는 앞으로 만들어질 대규모의 신경망 칩에서 이 구조가 유용하게 쓰여질 수 있음을 보인 것이다.

참 고 문 헌

[1] A Heteroassociative Memory Using Current-Mode MOS Analog VLSI Circuits K. A. Boahen, P. O. Pouliqen, A. G. Andreou, and R. E. Jenkins, IEEE Transactions on Circuts and Systems(May 1989).

[2] Analog Electronic Neural Network Circuits H. P. Graf and L. D. Jackel, IEEE Cicuits and Devices Magagine(July 1989).

[3] Implementing Neural Architectures Using Analog VLSI Circuits M. A. C. Maher, S. P. DeWeerth, M. A. Mahowald, and C. A. Mead, IEEE Transactions on Circuits and systems.

[4] A Programable Analig Neural Computer and Simulator P. Mueller, J Van der Spiegel, D. Blackman, T. chiu, T. Clare, J. Dao, C. Donharm, T.Hsieh, and M. Linaz, Advances in Neural Information Processing Systems.

[5] Neural Networks in CMOS A.Masaki, Y. Hirai, and M. Yamada, IEEE Cicuits and Devices Magagine.

[6] Hans P. Graf, Lawrence D. Jackel, and Wayne E. Hubbard. Vlsi Implementation of a neural network model. IEEE Computer, 21(3):41-49, 1988.

[7] A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning D. hammerstrom, Proceedings of the International Joint Conference on Neural Networks.

[8] Digital Systems for Artificial Neural Networks L. E. Atlas and Y. suzuki, IEEE Circuits and Devices Magazine.

[9] Mark Holler, Simon Tam, Hernan Castro, and Ronald Benson. An electrically trainable artificial neural network model with 10,240 floating gate synapses. In proceedings of the IJCNN, 1989.

[10] Nelson Morgan, editor. Artificial Neural Networks: Electronic Implementaions. Computer Society Press Technoligy Series and Computer Society press of the IEEE, Washington, D.C., 1990.

[11] VLSI Architectures for SAR Data Compression. H. Jeong, J. H. Park, H. Y. Ryu, J.

B. Kwon, and B. S. Koo EE Dept., POSTECH, Pohang 790-784, South Korea.

[12] Hong Jeong. Modeling and transformation of systolic network. Master's thesis, Massachusetts Institute of Technology, 1984.

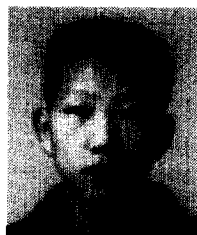
[13] T. J. Robnett and B. Charny. A high-speed systolic apeture radar processor. In IEEE International Conference on Acoustics, Speech, and Signal Processing, volume 4, page 357-360, Feb. 1992.

[14] Chales E. Leiserson, Flavio M. Rose, and Jaimes B. Saxe. Digital cicuit optimization. VLSI Memo No. 82-100, Apr. 1982.

[15] J.F.McDonald, et al., "The Trials of Wafer -Scale Integration," IEEE SPECTRUM, October, pp. 32-39, 1984.

[16] R.O.Carson, C.A.Neugebauer, "Future Trends in Wafer Scale Integration," Proceedings of IEEE, vol. 74, NO. 12, pp. 1741-1752, Dec. 1968.

[17] Y.Hirai et al., "A Digital Neuro-Chip with Unlimited Connectability for Large Scale Neural Networks," Proceedins of the IEEE and INNS IJCNN'89,vol.2,pp.163-169,1989.

[18] Paul Mueller et al., "A Programmable Analog Neural Computer And Simulator", Artificial Neural Networks, IEEE PRESS, pp. 218-224

이 윤 구

포항공대 학사 졸

정 홍

1973년 3월 서울대학교 전기공학과 학사
1977년 3월 한국과학원 전기 및 전자 공학과 석사
1979년 1월 경북대학교 전자공학과 조교
1980년 1월 경북대학교 전자공학과 전임강사
1982년 9월 MIT EECS 석사
1984년 6월 MIT EECS Engineer
1986년 6월 MIT EECS 박사
1988년 1월 포항공과대학교 전자전기공학과 조교수
1994년~현재 포항공과대학교 전자전기공학과
E-mail : yklee@hjmars.postech.ac.kr